Yet Another Compressed Cache: a Low Cost Yet Effective Compressed Cache

SOMAYEH SARDASHTI, University of Wisconsin-Madison
ANDRE SEZNEC, IRISA/INRIA
DAVID A WOOD, University of Wisconsin-Madison

RESEARCH REPORT
N° 8853

Feb 2016

Project-Team ALF

Yet Another Compressed Cache: a Low Cost Yet Effective Compressed Cache

SOMAYEH SARDASHTI, University of Wisconsin-Madison
ANDRE SEZNEC, IRISA/INRIA
DAVID A WOOD, University of Wisconsin-Madison

Project-team ALF

Research Report N° 8853— Feb 2016 —22 pages.

Abstract:

Cache memories play a critical role in bridging the latency, bandwidth, and energy gaps between cores and off-chip memory. However, caches frequently consume a significant fraction of a multicore chip's area, and thus account for a significant fraction of its cost. Compression has the potential to improve the effective capacity of a cache, providing the performance and energy benefits of a larger cache while using less area. The design of a compressed cache must address two important issues: i) a low-latency, low-overhead compression algorithm that can represent a fixed-size cache block using fewer bits and ii) a cache organization that can efficiently store the resulting variable-size compressed blocks. This paper focuses on the latter issue.

In this paper, we propose YACC (Yet Another Compressed Cache), a new compressed cache design that uses super-blocks to reduce tag overheads and variable-size blocks to reduce internal fragmentation, but eliminates two major sources of complexity in previous work—decoupled tag-data mapping and address skewing. YACC's cache layout is similar to conventional caches, eliminating the back-pointers used to maintain a decoupled tag-data mapping and the extra decoders used to implement skewed associativity. An additional advantage of YACC is that it enables modern replacement mechanisms, such as RRIP. For our benchmark set, YACC performs comparably to the recently-proposed Skewed Compressed Cache (SCC) [Sardashti et al. 2014], but with a simpler, more area efficient design without the complexity and overheads of skewing. Compared to a conventional uncompressed 8MB LLC, YACC improves performance by on average 8% and up to 26%, and reduces total energy by on average 6% and up to 20%. An 8MB YACC achieves approximately the same performance and energy improvements as a 16MB conventional cache at a much smaller silicon footprint, with 1.6% higher area than an 8MB conventional cache

Key-words: computer architecture, cache, compresed caches

RESEARCH CENTRE
BRETAGNE ATALANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex France

Yet Another Compressed Cache: a Low Cost Yet Effective Compressed Cache

Résumé : Nous présentons un nouveau modèle de cache compressé, ayant les même qualités que les caches compressés introduits récemment, mais une plus complexité matérielle réduite.

Mots clés : architecture de processeurs, mémoire cache, compression de cache.

Yet Another Compressed Cache:

a Low Cost Yet Effective Compressed Cache[1]

SOMAYEH SARDASHTI, University of Wisconsin-Madison
ANDRE SEZNEC, IRISA/INRIA
DAVID A WOOD, University of Wisconsin-Madison

Cache memories play a critical role in bridging the latency, bandwidth, and energy gaps between cores and off-chip memory. However, caches frequently consume a significant fraction of a multicore chip's area, and thus account for a significant fraction of its cost. Compression has the potential to improve the effective capacity of a cache, providing the performance and energy benefits of a larger cache while using less area. The design of a compressed cache must address two important issues: i) a low-latency, low-overhead compression algorithm that can represent a fixed-size cache block using fewer bits and ii) a cache organization that can efficiently store the resulting variable-size compressed blocks. This paper focuses on the latter issue.

In this paper, we propose YACC (Yet Another Compressed Cache), a new compressed cache design that uses super-blocks to reduce tag overheads and variable-size blocks to reduce internal fragmentation, but eliminates two major sources of complexity in previous work—decoupled tag-data mapping and address skewing. YACC's cache layout is similar to conventional caches, eliminating the back-pointers used to maintain a decoupled tag-data mapping and the extra decoders used to implement skewed associativity. An additional advantage of YACC is that it enables modern replacement mechanisms, such as RRIP. For our benchmark set, YACC performs comparably to the recently-proposed Skewed Compressed Cache (SCC) [Sardashti et al. 2014], but with a simpler, more area efficient design without the complexity and overheads of skewing. Compared to a conventional uncompressed 8MB LLC, YACC improves performance by on average 8% and up to 26%, and reduces total energy by on average 6% and up to 20%. An 8MB YACC achieves approximately the same performance and energy improvements as a 16MB conventional cache at a much smaller silicon footprint, with 1.6% higher area than an 8MB conventional cache.

• Computer systems organization → Architectures → Multicore architectures

Additional Key Words and Phrases: compression, cache design, energy efficiency, performance, multi-core systems

[1]This work is supported in part by the European Research Council Advanced Grant DAL No 267175 and the National Science Foundation (CNS-1117280, CCF-1218323, CNS-1302260, CCF-1438992, and CCF-1533885). The views expressed herein are not necessarily those of the NSF. Professor Wood has significant financial interests in AMD, Google, and Panasas. The authors would like to acknowledge Hamid Reza Ghasemi, members of the Multifacet research group for this helpful comments.

INTRODUCTION

Cache memories play an increasingly critical role in bridging the latency, bandwidth, and energy gaps between cores and main memory. However, caches frequently consume a significant fraction of a multicore chip's area, and thus account for a significant fraction of its cost. Compression has the potential to improve the effective capacity of a cache, providing the performance and energy benefits of a larger cache while using less area. Prior work has proposed compression algorithms suitable for last-level cache (LLC) designs, with low latencies and efficient hardware implementations [Alameldeen, and D. Wood. 2004; Ziv et al. 1977; Ziv et al. 1978; Huffman. 1952; Vitter. 1987; Pekhimenko et al. 2012]. Other work has focused on the organization of a compressed cache, which must efficiently compact and retrieve variable-size compressed cache blocks [Alameldeen, and D. Wood. 2004; Hallnor, and Reinhardt. 2005; Kim et al. 2002; Sardashti and Wood. 2013; Sardashti et al. 2014].

A compressed cache organization must provide additional tags, support variable-size data allocation, and maintain the mappings between tags and data. The additional tags allow a compressed cache to hold more compressed than uncompressed blocks, but incurs area overheads. Variable-size data allocation allows compressed blocks to be stored in the cache with low internal fragmentation, but may require expensive re-compaction when a block changes and requires more space. And the combination of additional tags and variable-size blocks makes it challenging to maintain the mapping between tags and data. Finally, a compressed cache would ideally enable advanced LLC replacement policies, such as RRIP [Jaleel et al. 2010].

Recent work on Decoupled Compressed Cache (DCC) [Sardashti and Wood. 2013] demonstrated that super-block tags and sub-block allocation could reduce tag overhead and eliminate re-compaction overheads, but with the additional area and complexity of backward pointers to maintain the tag-data mapping. DCC also adds complexity to cache replacements due to its sub-block data allocation. Skewed Compressed Cache (SCC) [Sardashti et al. 2014] also uses super-block tags, but eliminates the area of DCC's backward pointers using a skewed-associative lookup [Seznec. 1993; Seznec. 2004.] and direct tag-data mapping. SCC also simplifies cache replacement by always replacing an entire super-block, rather than individual blocks. However, skewed associativity has not found wide-spread adoption by industry, in part due to the need for a separate address decoder for each tag way. Skewing also eliminates the conventional notion of a cache set, making it difficult or impossible to exploit many of the modern LLC cache replacement policies that have been proposed in the past decade [Jaleel et al. 2010].

In this paper, we propose YACC (Yet Another Compressed Cache). Similar to DCC and SCC, YACC is agnostic to the compression algorithm and uses super-block tags to provide an efficient compressed cache organization. YACC exploits the properties of compression locality (i.e., neighboring blocks tend to have similar compressibility) and spatial locality (i.e., neighboring blocks tend to co-exist in the cache). Similar to SCC, YACC compacts neighboring blocks with similar compression ratios in one data entry (i.e., 64 bytes) and tracks them with a sparse super-block tag. Unlike SCC, YACC uses a conventional non-skewed tag-data mapping, allowing it to allocate a compressed super-block in any way of a conventional set. YACC's simple, conventional tag mapping allows designers to implement the whole spectrum of recently-proposed LLC replacement policies.

On our set of benchmarks, YACC improves system performance and energy by on average 8% and 6%, respectively, and up to 26% and 20%, respectively, compared to a regular uncompressed 8MB cache. Similar to DCC or SCC, YACC achieves comparable performance and energy as a conventional cache of twice the size, using far less area. Compared to DCC, YACC uses less area and a simpler access path, by eliminating the backward pointers, and a much simpler super-block-only replacement mechanism. Compared to SCC, YACC eliminates the extra area and complexity of skewing. Furthermore, unlike either DCC or SCC, YACC can also use modern replacement policies, such as RRIP [Jaleel et al. 2010], further reducing cache design complexity.

This paper is organized as follows. We discuss basics of compressed caching and related work in Section 2. Section 3 presents our proposal, YACC. Section 4 explains our simulation infrastructure and workloads. In Section 5, we present our evaluations. Finally, Section 6 concludes the paper.

Table 1: Compressed Cache Taxonomy

	FixedC 00	VSC 0	IIC-C 0	DCC 00	SCC 0	YACC [new]
Super-Block Tags	✗	✗	✗	✓	✓	✓
Single Tag Decoder	✓	✓	✓	✓	✗	✓
Direct Tag-Data Mapping	✓	✗	✗	✗	✓	✓
Variable-Size Blocks	✗	✓	✓	✓	✓	✓
Re-compaction Free	✓	✗	✓	✓	✓	✓
Flexible Replacement Policies	✓	✓	✗	✗	✗	✓

BACKGROUND AND RELATED WORK

There are several cache structure proposals on using compression in on-chip caches. Earlier works [Kim et al. 2002; Lee et al. 2000] limit the benefits of compression mainly by limiting the number of tags, and not tightly packing variable size compressed blocks. Recent proposal, DCC 0, proposes different mechanisms that address many of these issues, but at some extra costs and complexity due to adding one level of indirection (i.e., backward pointers), and separately managing block and super-block replacements. The more recent work, SCC, addresses these remaining issues in DCC, eliminating DCC's backward pointers and simplifying cache replacement mechanism. On the other hand, SCC adds the complexity of skewing, complicating the tag array design, and limiting the choice of replacement policy. In this section, we summarize the basics of compressed cache designs, and discuss these previous works and their trade-offs in more detail. In the next section, we explain our proposal (YACC) that addresses these remaining issues with SCC, achieving the same benefits from compression with a much simpler design.

Basics of Compressed Caches

In general for a compressed cache design, designers pick one/more compression algorithms to compress blocks, and use a compaction mechanism to manage compressed blocks in the cache. There are several compression algorithms proposed in the past [Alameldeen, and D. Wood. 2004; Pekhimenko et al. 2012; Chen et al. 2010]. At cache level, the complexity and overhead of the algorithm matter the most. As decompression latency is on the critical path, algorithms with low decompression latency, but yet good compressibility for small blocks are suitable for cache compression. C-PACK [Chen et al. 2010] has been shown to have low hardware overheads and decompression latency, with fairly high compression ratio [Chen et al. 2010; Sardashti and Wood. 2013]. C-PACK is designed specifically for hardware-based cache compression. It detects and compresses frequently appearing words (such as zero words) to fewer bits. In addition, it also uses a small dictionary (e.g., 16 4-byte entries) to compress other frequently appearing patterns. Overall it has a decompression latency of 9 cycles. There are other low-overheads algorithms appropriate for cache compression, such as BDI and FPC. They provide lower decompression latency, but usually at lower compressibility [Sardashti and Wood. 2013]. Our work is independent of compression algorithms. In this study, we use C-PACK+Z algorithm [Sardashti and Wood. 2013] that is a variation of C-PACK algorithm with support to detect zero blocks.

Given a compression algorithm, a compaction mechanism is the key in designing an efficient compressed cache. In this work, we focus on providing a simple yet effective compaction mechanism. A compaction mechanism needs to address the following issues: (1) how to provide extra tags in the tag array at a reasonable storage overhead? Since a compressed cache can fit more blocks in compressed format in the same space as a regular uncompressed cache, it needs to provide some extra tags to track

them; and (2) how to allocate compressed blocks and map them with the tags (tags-data mapping)? Since compressed blocks could have different sizes, the cache needs a mechanism to locate a block in the data array given a matched tag.

States of the Art Compressed Caches

There are different trade-offs in current compressed cache designs. We summarize them in Table 1.

Super-Block vs. Block Tags: In general there are two main approaches to provide extra tags: simply increasing the number of tags, or managing the tag array at super-block level. Several designs [Alameldeen, and D. Wood. 2004; Kim et al. 2002; Baek et al. 2013] simply increase the number of tags, for example doubling tag array size (e.g., 32 tags per set for a 16-way associative cache). Further increasing the number of tags would increase tag area overhead. On the other hand, previous work SCC and DCC propose using super-block tags. Super-block tags track multiple neighboring blocks (e.g., up to 4 blocks) with one super-block tag [Sardashti and Wood. 2013], reducing the tag overhead.

Direct vs. Decoupled Tag-Data Mapping: Given a matched tag, the next step is to find and read out/write data from/to the data array. We can categorize existing techniques into: direct tag-data mapping, and decoupled tag-data mapping. Direct mapping associates a tag with a particular data entry, similar to regular uncompressed caches. Using direct tag mapping usually makes compressed cache designs simpler, as by matching a tag, we can simply locate the block in the data array without requiring any extra metadata. Decoupled mapping techniques need to add another level of indirection, and so extra metadata to locate a block in the data array. Using direct tag mapping usually makes compressed cache designs simpler, as by matching a tag, we can simply locate the block in the data array without requiring any extra metadata. Decoupled mapping techniques need to add another level of indirection, and so extra metadata to locate a block in the data array.

Variable-Size vs. Fixed-Sized Compressed Blocks: Early design proposals with direct tag-data mapping (FixedC [Kim et al. 2002; Lee et al. 2000]) were limiting compression benefits by only allocating blocks as fixed block sizes (either half a block, or a full block), even if blocks were highly compressible. As a result, several decoupled mapping techniques (such as in VSC[Alameldeen, and D. Wood. 2004], IIC-C and DCC[Sardashti and Wood. 2013]) were proposed to store variable size compressed blocks. Decoupled mapping allows flexible allocation of a compressed block in the cache, meaning that there is no one-to-one corresponding between a matched tag and the data entry in the data array. By tightly packing variable size compressed blocks, decoupled mapping can reduce internal fragmentation, and improve overall cache effective capacity.

Re-Compaction Overhead with Variable-Size Compression: In addition to extra metadata, VSC needs to frequently re-compact blocks on updates when block sizes grow (i.e., re-compaction). As updates happen frequently, re-compaction could have significant overheads on cache dynamic energy. DCC and IIC-C provide re-compaction free decoupled mapping but introduce extra metadata. DCC allows sub-blocks of a compressed block to be stored anywhere in a cache set. It further requires backward pointers (i.e., few bits per sub-block in the data array representing its owner) to locate a block. Similarly, IIC-C requires forward pointers (i.e., indexes to the cache sets storing sub-blocks of a given block) to find sub-blocks of a block anywhere in a fully-associative cache.

Among previously proposed designs, SCC [Sardashti et al. 2014] provides so far the best balance among these trade-offs, getting the benefits of super-block tags and simple direct tag-data mapping, while supporting variable size compression. SCC also addresses the major remaining issues with DCC, namely, the complex replacement automaton and the extra metadata associated with block location. DCC complicates replacement mechanism as it manages replacing super-blocks and blocks separately. SCC associates a super-block tag to its corresponding data entry. It compresses blocks, and stores multiple blocks with the same compressibility and from the same super-block into a fixed size data entry (e.g. 64 bytes). In this way, using a direct tag-data mapping, it does not need any extra metadata to locate a block. However, SCC limits the cache ways a block could be mapped to. For example, in a 16-way cache, for a given compression factor, a block can only be stored in 4 ways. So, SCC reduces

effective associativity of the cache. For some workloads, this was partly compensated by extra skewing on the groups of ways (i.e., indexing each way differently). However, SCC still suffers from a few limitations that we list below.

Single vs. Multiple Tag Decoder: In a regular cache, when looking up a block, we index a set of tag arrays using a single decoder, and check all the tags in that set for a possible match. Similarly, in most previous compressed cache designs, a single tag decoder can be used. In SCC, however, each way is indexed differently using skewing. Therefore, instead of one decoder for the tag array, SCC needs one decoder per cache way, i.e., the tag arrays must be built with one physical array per way. This would complicate tag array design and layout.

Flexible Cache Replacement Policies: Ideally a compressed cache should be able to use any replacement policy. However, compressed cache designs could limit or complicate replacement policies. DCC, for example, complicate cache replacement logic, a new cache allocate could result in multiple super-block and block replacements. SCC also inherits the replacement issues associated with skewing. Skewing would limit the choice of replacement policy in use. As a skewed based cache indexes each cache way using a different index, there is no notion of a fix cache set. A replacement candidate can be in a different set in different cache ways. Thus, it is not possible to use skewed caches (including SCC) with modern replacement policies (e.g., RRIP) that have been recently proposed for conventional set-associative LLC. Skewed caches replacement policies should rely on more complex replacements scheme such as Zcache [Sanchez, and Kozyrakis. 2010].

Difficult to Understand Behavior: In addition, skewed caches have not been so far adopted in last-level caches despite their intrinsic advantages to amortize peaks of conflict misses [Seznec. 1993]. This reluctance partly comes from the application community that expresses interrogations on the predictability of the behavior of skewed caches.

The YACC cache described in the next section addresses these remaining issues with SCC. YACC provides similar high benefits with compression, while simplifying the cache design. YACC removes the overheads of skewing in SCC, simplifying the tag array and allowing the whole spectrum of replacement policies.

YACC

YACC Architecture

Figure 1 shows a high level overview of YACC architecture. For the sake of simplicity, throughout this paper, the (uncompressed) data block size is 64 bytes. For this example, we are using an 8-way associative cache. To keep the benefits of previous works, YACC also uses super-block tags to reduce tag area overhead. YACC keeps tag-data mapping simple by using a direct tag-data mapping. Each tag indicates which blocks are present in its corresponding data entry. YACC exploits both spatial locality (i.e., there are many neighboring blocks in the cache) and compression locality (i.e., neighboring blocks tend to have similar compressibility). Similar to SCC, YACC stores neighboring blocks in one data entry if they have similar compressibility, and could fit in one data entry. It then tracks blocks stored in one data entry with the corresponding sparse super-block tag. Unlike SCC, YACC does not skew the cache, and so does not limit the possible location of a block in the set. YACC simply indexes tag and data arrays with the super-block index. It can store a block in any way in that indexed set. We will next discuss YACC design in more detail.

How to provide extra tags?

When compressing blocks, we can fit more blocks in the cache. So, we need to provide more tags to track them. To keep tag overhead low, YACC uses sparse super-block tags [Sardashti et al. 2014] to represent the blocks that are compressed in the corresponding data. For example, blocks E,F,G,H of SB2 are each compressible to one 16-byte sub-block. Thus, YACC stores them together in one data entry (i.e., 64 bytes). Blocks M and P of SB4 are each compressible only to 32-byte, Thus, YACC stores them in a single data entry while blocks O and N are not compressible and occupy each a data entry.

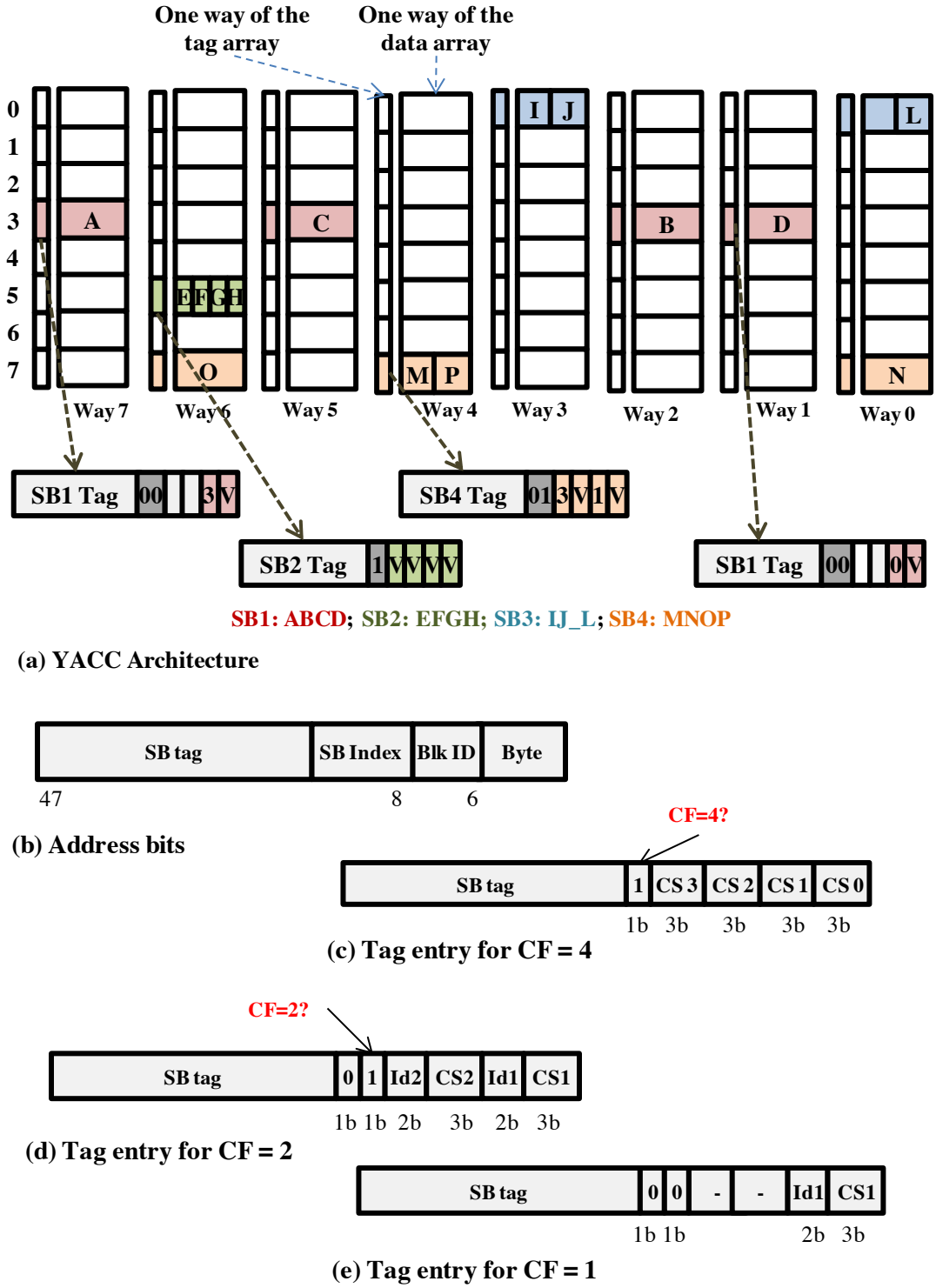

Fig. 1. YACC Architecture (a): in an 8-way associative cache, YACC associates one tag entry (sparse super-block tag) per data entry (i.e., 64 bytes). Each tag entry tracks up to 4 neighboring blocks (b), if they have similar compressibility and could fit in one 64-byte data entry (e.g., SB2). If neighboring blocks have different levels of compressibility, they would be allocated in different data entries in the same set (e.g., in SB4, M and P are stored together, while N and O are allocated separately). For each data entry, the corresponding tag indicates which blocks are present (c, d, e).

YACC tracks them by the corresponding sparse super-block tag entry. In this example, the tag indicates that all four blocks are valid, and belong to SB2. By tracking super-blocks, YACC can track up to four times more blocks with small area overhead. Each tag entry is slightly larger than a regular tag

entry as it needs to store coherency states for all blocks of a super-block. In practice 8 extra bits over a conventional tag are sufficient.
Figure 1 also shows the structure of YACC tag entries. Each tag (i.e., sparse super-block tags) would indicate which blocks of a super-block are stored in the corresponding data entry. Thus, each tag indicates the super-block tag address, compressibility of these blocks, and per-block coherency/valid states. To represent this information with minimum bits, in addition to super-block tag address, YACC uses one bit to indicate if the blocks have compression factor (CF) of 4 (i.e., compressible to 16 bytes). If so, the next bits present coherence states (CS) of all four blocks (i.e., CS3-0), as shown in Figure 1 (c). For example, blocks E, F, G, and H of SB2, for which CF is 4, are all stored in one data entry, so their corresponding tag reflects that (Figure 1 (a)). If CF is not 4, this bit is set to zero, and the next bit will be used to show how the blocks are compressible. If the blocks are compressible to a factor of two (i.e., compressible to 32 bytes), we can have maximum of two blocks in one data entry, each compressible to half a block. Thus, we use the next bits in the tag entry to present block id and coherence states of each of those blocks (Figure 1 (d)). Finally, if CF is one (i.e., uncompressible), we can store one uncompressed block in the corresponding data entry, so YACC will present its block id and coherence state in the tag (Figure 1 (e)). For example, for block A or block D, which are not compressible, the corresponding tags indicate their valid coherence state, and compression factor, as shown in Figure 1 (a).

How to allocate compressed blocks?

YACC uses a direct tag-data mapping to keep compressed block allocation simple. It compresses blocks into a power of two numbers of sub-blocks (e.g., one, two, or four 16-byte sub-blocks). It then stores neighboring blocks in one data entry if they could fit (e.g., blocks E,F,G,H of SB2). Otherwise, it stores them in one or more other data entries in the same data set. For example, blocks I, J, and L from SB3 are each compressible to half a block. YACC fits I and J in one data entry (way 3 of set 0), and stores L in a separate data entry in the same set (way 0 of set 0). Later if we access block K from SB3, YACC could compress and store it in the same data entry as L if it could fit.
In case the blocks belonging to the same superblock are not compressible, such as blocks A, B, C, D of SB1, YACC stores these blocks separately in different data entries (in way 7, 5, 2, and 1) in the same cache set (set #3). In this way, these blocks compete on the same set in the cache. Thus, an application featuring low data compressibility and high spatial locality, might suffer from a limited visible associativity, as we will discuss in the evaluation section.
YACC mapping is based on SCC as YACC also stores blocks of a single super-block into one data entry, if possible, and tracks them with sparse super-block tags. On the other hand, YACC mapping is different from SCC in multiple ways as it aims to achieve similar benefits with a simpler design:

No cache skewing: SCC mapping is complex because of skewing. In SCC, a block is mapped to particular cache ways depending on its address and compressibility. For example, in a 16-way associative cache, a block could only be stored in 4 ways. SCC further employs skewing to compensate for the limited associativity.

Figure 2 shows the physical view of SCC vs. YACC. On a lookup, where the block size is unknown, SCC checks all possible positions of the block (8 in an 8-way associative cache). SCC skews each cache way with a different hash function. For example in Figure 2 (b), the colored blocks (yellow and red) show the tags being checked on a lookup. Note that because of skewing each tag is in a different row. Therefore, SCC requires eight decoders on the tag array, one per cache way. This would complicate tag array design, area, and possibly layout and routing. On the other hand, YACC stores a block in any cache way in a given cache set (indexed by super-block index bits). For example, in Figure 2 (a), on a lookup, all tags of set 1 (in yellow and red) will be checked for a possible hit. In this way, YACC does not limit cache associativity, and similar to a regular cache, only requires one decoder for the whole tag array. Note that the extra decoders in SCC are only needed for the tag array and not the data array. In both SCC and YACC, when a tag matches, we will then index and access the corresponding data entry in the data array (colored in red in Figure 2). In addition to complicating tag array design, skewing would also

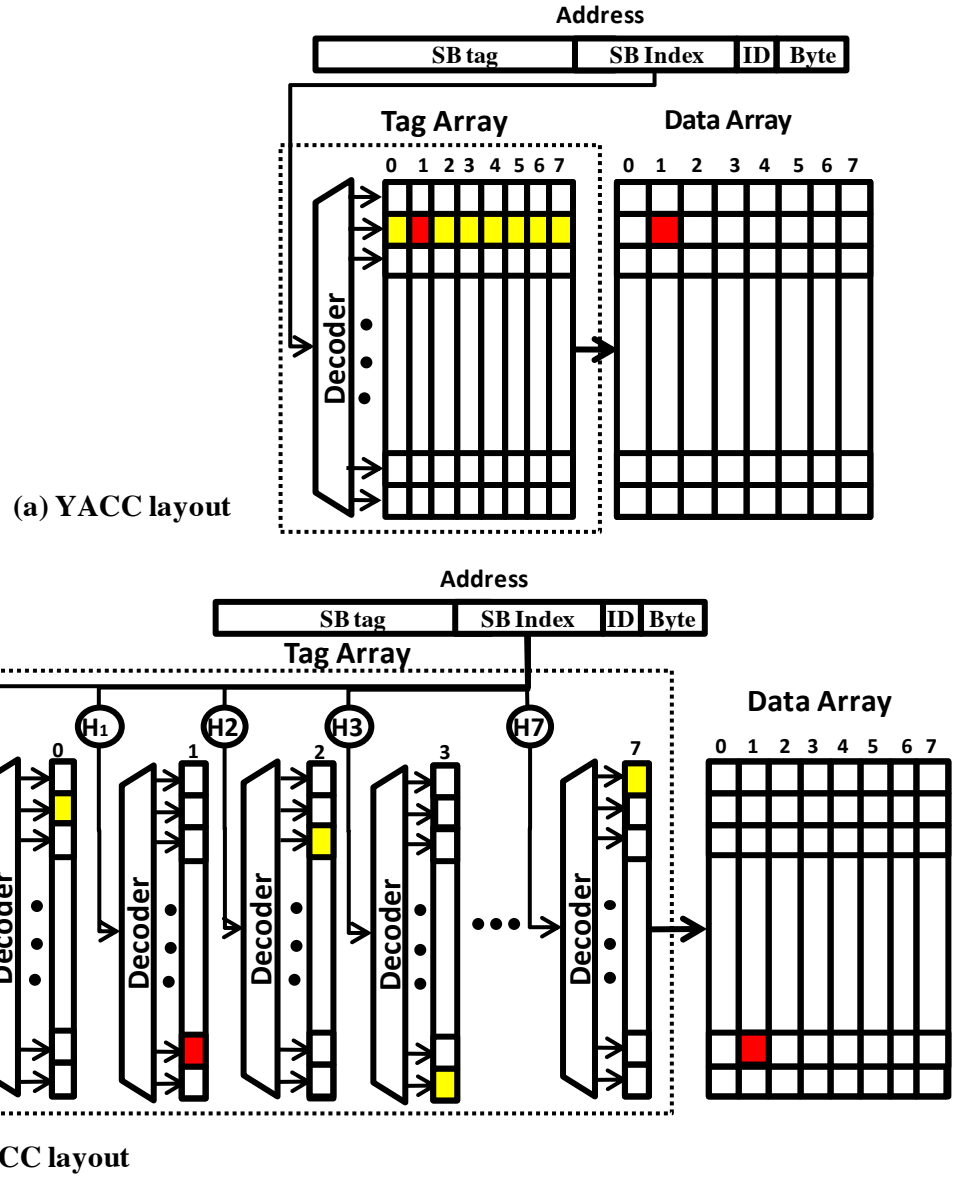

Fig. 2. Cache physical layout with YACC (a), and SCC (b)

limit the choice of replacement policy. In SCC, for any block, SCC could map it to a different set of rows.

Thus, there is no fixed notion of a cache set. On the other hand, YACC can use any replacement policy, such as RRIP [Jaleel et al. 2010].

In-place block expansion: YACC allows in-place expansion of a block on a write-back from lower cache levels, if the block size grows and it is the only resident of a data entry. For example, if block L's size grows, so that it becomes uncompressible (requiring 64 bytes), YACC will store it in the same data entry. It only changes its status in the corresponding tag. SCC will need to invalidate, and reallocate that block to a different data entry on a block update.

Storing non-adjacent blocks together: In order to compact more blocks, YACC does not necessarily compact blocks of a super-block in the same order. For example, blocks M and P from SB4 are compressible to 32 bytes each. Although they are not contiguous neighbors, YACC would still pack them together. Previous work, SCC, does not support this mode as it stores blocks in strict order. In a similar situation, SCC would map these blocks to different entry, and would allocate a data entry for each. For example, SCC would allocate M and P in different data entries, using 128 bytes (2*64 bytes) instead of 64 bytes (2*32 bytes) in YACC. In order to locate these blocks, YACC encodes the tag entry differently when CF is 2 (shown in Figure 1). The tag entry includes block id (2 bits) and coherence state (3 bits) of the blocks stored into the lower or the upper half of the corresponding data entry. For example, the tag entry in way #4 of set #7 indicates that block #3 of SB4 (block M) and block #1 of SB4 (block P) are stored in upper and lower halves of the corresponding data entry, respectively.

YACC Cache Operations

Cache Read

Figure 3 shows a block diagram of main cache operations in YACC. On a cache read, YACC indexes a set of tag array using super-block index bits from the address. For example, in Figure 1, to read block A of SB1, YACC will index set #3. It then checks all tag entries in that set for a possible hit. A cache hit occurs if the tag address matches, and the corresponding coherence state is valid. For example, YACC finds block A in way #7 of set #3, as the super-block tag address in that tag entry matches with SB1 tag address, and the tag entry indicate the block A is valid (block #3 exists in Valid state). Note that since YACC maps all blocks of a super-block to the same set, it is possible to have more than one tag entry matching the super-block tag address, but only a single corresponding valid tag match. For example, the tag entry in way #1 of set #3 also tracks part of SB1 (i.e., block D), so it also has the same super-block tag address but block A is invalid. On a read hit, YACC would read out and decompress the corresponding sub-blocks from the data array. On a cache miss, YACC would allocate the block in the cache.

Cache Allocate and Cache Replacement

On a cache allocate (e.g., caused by a cache miss), a cache write or write-back, YACC first compresses the block. It then indexes a set of the tag array using the super-block index bits from the address. YACC then checks for a tag entry with the same super-block tag address and compression factor. If so, it then checks to see if the block can fit in the corresponding data entry. For example, if a block has CF of 2, and there is a tag with the same CF and super-block tag address matching, YACC will allocate the block there if there is only one other valid block in that entry. For example, we can allocate block K (from SB3) in the same entry as block L, if K is also compressible to half.

On a write-back (or update) to an existing block, if the block size grows, YACC might need to invalidate the previous version of this block before re-allocating it. If the block is the only block in that entry, such as block L, YACC will not invalidate or re-allocate it. It simply stores the block in the same entry, and only updates the tag. Otherwise, if it does not fit in its previous entry, it would invalidate and allocate it in the cache as just explained.

In case there is no matching tag entry with enough space for the accessing block, YACC needs to replace a victim sparse super-block first before allocating this block. Finding the victim tag is straightforward, and basically similar to a regular cache. Depending on the replacement policy, it finds the victim tag, and evicts the blocks resident in its entry. For example, if YACC picks way #4 of set #7 as victim, it would evicts blocks M and P of SB4, and free that entry. Note that other blocks of SB4 (i.e., blocks N, and O), which are not resident of this particular entry, will still stay in the cache.

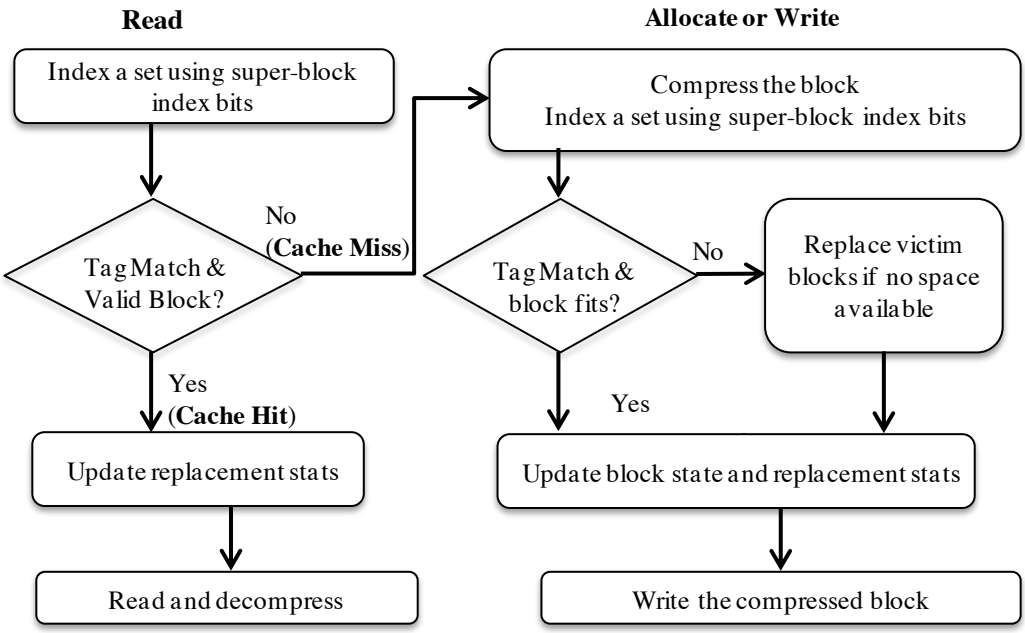

Fig. 2. YACC cache operations.

Methodology

Table 2. Simulation Parameters

Processors	8, 3.2 GHz, 4-wide issue, out-of-order
L1 Caches	32 KB 8-way split, 2 cycles
L2 Caches	256 KB 8-way, 10 cycles
L3 Cache	8 MB 16-way, 8 banks, 27 cycles
Memory	4GB, 16 Banks, 800 MHz DDR3.
Block Size	64 bytes
Super-Block Size	4-block super-blocks
Sub-block Size	16 bytes

To evaluate YACC, we use the same evaluation framework that was used to evaluate SCC and DCC. We use full-system cycle-accurate GEMS simulator [Martin et al. 2005]. We model YACC with an 8-core multicore system with OOO cores, per-core private L1 and L2 caches, and one shared last level cache (L3). We implement YACC and other compressed caches at the L3. Table 2 shows the main parameters. We use 64-byte cache block sizes. For YACC, SCC [Sardashti et al. 2014], and DCC [Sardashti and Wood. 2013], we use 4-block super-blocks (each tag tracks 1-4 neighbors), and 16-byte sub-blocks (i.e., each block compress to 0-4 sub-blocks). We also use CACTI 6.5 [CACTI] to model area and power at 32nm.

We use several applications with different characteristics from SPEC OMP [Aslot et al. 2001], PARSEC [Bienia and Li. 2009], commercial workloads [5], and SPEC CPU 2006. From SPEC CPU 2006 benchmarks, we run mixes (mix1 – mix8) of multi-programmed workloads from memory-bound and

compute-bound. For example, for omnetpp-lbm, we run four copies of each benchmark. Table 3 shows our applications. We classify these workloads into: low memory intensive, medium memory intensive, and high memory intensive based on their LLC MPKI (Misses per Kilo executed Instructions) for the Baseline configuration (a regular uncompressed LLC). We classify a workload as low memory intensive If LLC MPKI is lower than one, as medium memory intensive if LLC MPKI is between one and five, and as high memory intensive if MPKI is over five. We run each workload for approximately 500M instructions with warmed up caches. To address workload variability, we simulate each workload for a fixed number of work units (e.g., transactions) and report the average over multiple runs [Alameldeen, and D. Wood. 2003]. Table 3 also shows the compression ratio (original size/compressed size) for each workload using C-PACK+Z algorithm [Sardashti and Wood. 2013; Chen et al. 2010].
We evaluate the following configurations for LLC:

- **Baseline** is a conventional uncompressed 8-way 8MB LLC.
- **2X Baseline** is a conventional 32-way 16MB LLC.
- **DCC** models Decoupled Compressed Cache [Sardashti and Wood. 2013] with 4-block super-blocks, and 16-byte sub-blocks. We use a LRU-based replacement algorithm for both super-block and block level replacements.
- **SCC** models Skewed Compressed Cache [Sardashti et al. 2014] with 4-block super-blocks, and 16-byte sub-blocks. We use a LRU-based replacement algorithm for super-block replacements.
- **YACC** models our proposal with 4-block super-blocks, and 16-byte sub-blocks. We also use a LRU-based replacement algorithm for super-block replacements.
- Baseline RRIP is also an uncompressed 8-way 8MB LLC, but we use RRIP for replacement policy [Jaleel et al. 2010].
- **YACC RRIP** is similar to YACC (or YACC LRU) configuration, but we use RRIP for replacement policy [Jaleel et al. 2010].

Evaluation

Hardware Overheads

Table 4 shows the area overhead of YACC and state of the art DCC and SCC. We assume a 48-bit physical address space. Compressed caches use the same data array space, but usually increase tag array space to track more blocks. In Table 4, we show the number of bits needed per set in a 16-way cache. We categorize tag space into bits needed to represent tags (e.g., tag addresses and LRU information), extra metadata needed for coherence information, and extra metadata for compression (e.g., compression factor).
State of the art DCC uses super-block tags to track more blocks at lower tag overhead. It uses the same number of tags, but each tracks up to 4 blocks of a super-block. The tags use fewer bits for the matching address (27-bit super-block tag address). DCC keeps LRU information separately per super-blocks (4 bits to find the least recently used super-block) and blocks (6 bits to find the least recently used block). Since DCC can fit up to 4 times more blocks in the same space, it keeps 4 times more coherence state (3 bits assuming MOESI). It also keeps one valid bit per super block. DCC decouples tag-data mapping, requiring extra metadata to hold the backward pointers that identify a block's location. DCC keeps one 6-bit backward pointer entry (BPE) per sub-block in a set. In addition, it stores one bit in the tag per block showing if the block is compressible or not. Overall, DCC more than doubles tag area, incurring about 6.7% area overhead on total LLC area (tag array and data array).
SCC cuts down on these extra tag bits. SCC keeps LRU information for super-blocks only, and completely eliminates extra compression metadata. In this way, SCC increases tag array area by about 24%, and total LLC area by only 1.5%. Here we are only counting the overheads in term of extra bits stored. SCC, however, requires 16 tag decoders instead of one, which we are not counting here.

In terms of tag and metadata bits stored per set, YACC is very similar to SCC. The only difference is that it keeps one extra bit per super-block, representing if the blocks are compressible to a factor of 4 (CF=4?). Compared to a conventional cache, YACC uses only 8 extra bits per tag entry. Unlike SCC,

Table 2. Compressed Caches Area Overhead

	Baseline	DCC	SCC	YACC
Tags per Set (bits)	16x29+4	16x27+4+6	16x27+4	16x27+4
Coherence Metadata per Set (bits)	16x3	16x4x3+16x1	16x4x3	16x4x3+16x1
Compression Metadata per Set (bits)	0	16x4x6+16x4x1	0	0
Total LLC Tag Array Overhead (%)	**0**	**113%**	**25%**	**28%**
Total LLC Overhead (%)	**0**	**6.7%**	**1.5%**	**1.6%**

YACC does not change tag array layout, it requires only one tag decoder and no hash function hardware to address the cache.

Cache Effective Capacity

In general compressed caches increase cache utilization by fitting more blocks in compressed format in the same space as an uncompressed cache. Figure 4 shows the effective capacity of different designs normalized to Baseline. To calculate the effective capacity of a cache, we count the number of valid blocks in the cache when allocating a new block, and report the average number over all counts. In

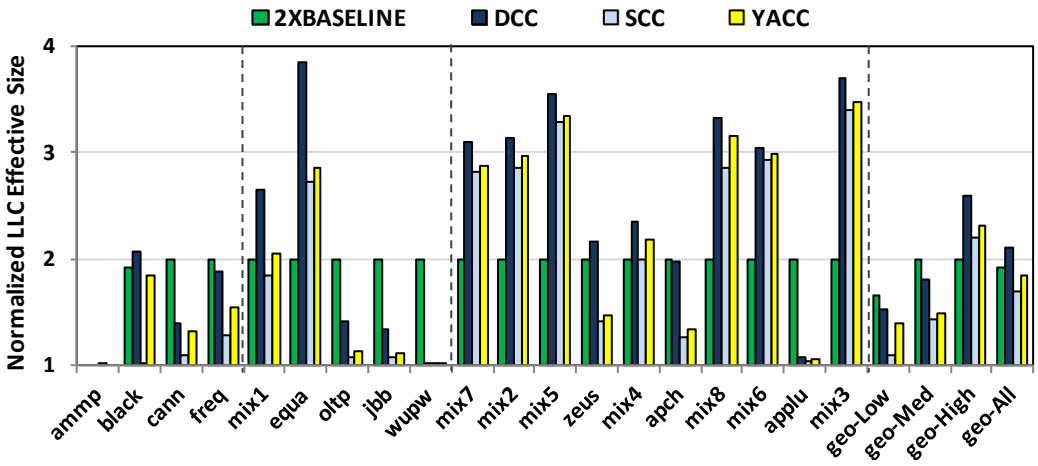

Fig. 3. Normalized LLC effective size

Figure 4, we report the effective capacity of each configuration, normalized to the Baseline. For an ideal compressed cache, the effective capacity should be the same as average compression ratio for each

benchmark. However, low memory intensive workloads with even good compressibility, such as ammp, do not have that large working set. That is why even when doubling the cache size (2X BASELINE) these workloads cannot use the whole cache (i.e., average normalized capacity of 1.6 with 2X BASELINE).

By packing compressed blocks, YACC improves effective capacity by up to 3.5 times for mix3 and on average by 84%. YACC on average improves cache effective capacity similar to a 2X BASELINE, while it has almost half area. Among our workloads, high memory intensive workloads benefit the most from YACC, and in general compression. YACC increases cache capacity more than twice for these workloads.

Although similar to SCC, YACC compacts neighboring blocks with similar compressibility, it improves effective capacity over YACC. YACC proposes in-place expansion, avoiding extra replacements when a block is the only resident in one data entry. It also packs two non-contiguous neighbors (like M and P in Figure 1) with CF of two in one entry. Thus, due to these optimizations, YACC achieves better effective capacity over SCC.

Among previous work, DCC provides higher normalized effective capacity than SCC and YACC. In SCC and YACC, only neighbors with similar compressibility share the space. In DCC, however, blocks can be stored anywhere in the cache, so the space freed by compressing one block can be used to store a non-neighboring block. Thus, overall, DCC provides the highest effective capacity, but at the cost of a more than four times the area overhead than YACC and SCC and more complex data access path.

Table 3. Applications

Application	**Compression Ratio**	**Baseline LLC MPKI**	**Category**
ammp	3.0	0.01	Low Mem Intensive
Blackscholes	4.0	0.13	
canneal	2.8	0.51	
freqmine	3.3	0.65	
bzip2 (mix1)	4.0	1.7	Medium Mem Intensive
equake	5.0	2.2	
oltp	2.0	2.3	
jbb	2.6	2.7	
wupwise	1.3	4.3	
gcc-omnetpp-mcf-bwaves-lbm-milc-cactus-bzip (mix7)	3.9	8.4	High Mem Intensive
libquantum-bzip2 (mix2)	3.7	9.3	
astar-bwaves (mix5)	3.8	9.3	
zeus	2.9	9.3	
gcc-166 (mix4)	4.2	10.1	
apache	2.8	10.6	
omnetpp-lbm(mix8)	4.7	11.2	
cactus-mcf-milc-bwaves (mix6)	4.8	13.4	
applu	1.7	25.9	
libquantum(mix3)	4.0	43.9	

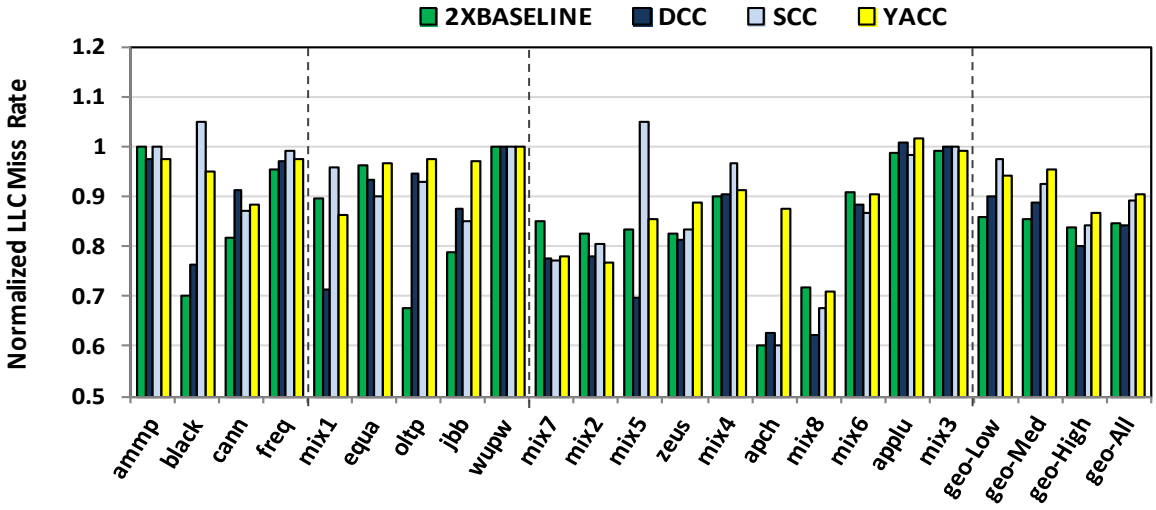

Fig. 4. Normalized LLC miss rate (MPKI). Applications are ordered based on their Basline MPKI.

Cache Miss Rate

By improving cache effective capacity, compressed caches tend to reduce cache miss rate. Figure 5 shows the LLC MPKI (Misses per Kilo executed Instructions) for different cache designs. When doubling the cache size, 2X Baseline improves LLC MPKI by on average 15%, and up to 40% for apache. However, these benefits come at twice larger LLC area, which is already one of the largest on-chip components.

YACC improves LLC MPKI by compressing blocks. It achieves most of the benefits of 2x Baseline, with about half area. YACC improves LLC MPKI by about 10% on average, and up to 30%. Among previous works, YACC performs similar to SCC. SCC uses compression, but limits cache effective associativity by only mapping a block into 4 out of 16 cache ways. On the other hand, it employs skewing to compensate for possible loss of associativity. Thus, for some workloads, such as apache, skewing combined with compression can improve overall miss rate, achieving lower LLC MPKI than YACC. While for some others, such as mix5, skewing would not compensate lower associativity in SCC. Compared to DCC, DCC lowers MPKI more than YACC and SCC, but at higher design complexity and overheads. Also note that on a few applications (applu, mix3, freq), doubling the cache size has virtually no impact on the miss rate, therefore a large compression factor on mix3 does not significantly help to reduce the misprediction rate.

System Performance and Energy

By improving cache effective capacity and lowering cache miss rate, compressed caches can improve system performance and energy. Figure 6 and Figure 8 show system performance and energy of different cache designs. We report total energy of cores, caches, on-chip network, and main memory, including both leakage and dynamic energy.

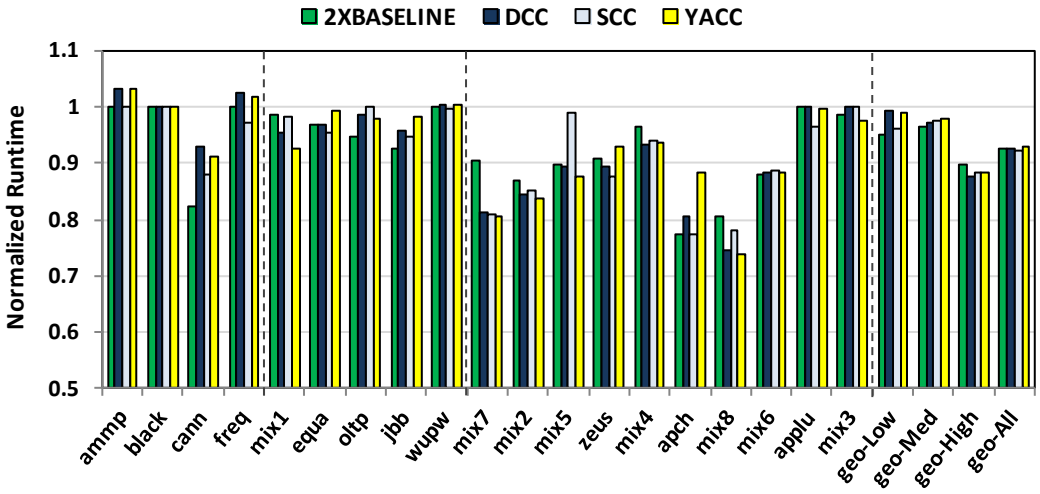

Fig. 5. Normalized performance

YACC improves performance by up to 26%, and on average 8%. It similarly lowers total energy by up to 20% and on average 6% by reducing the number of accesses to main memory and lowering the runtime. Overall, cache sensitive applications, for which miss rate reduces when increasing cache size, benefit the most from compression. Among our evaluated workloads, many applications from medium and high memory intensive category, such as jbb, apache, and mix8, benefit the most from YACC. On the other hand, cache insensitive workloads, which include low memory intensive workloads as well as some with very high miss rate would not benefit from YACC. For example, libquantum (mix3), which has about 43 MPKI, does not benefit from compression despite its high compressibility.

YACC achieves similar performance and energy benefits as 2x Baseline, and previous works, SCC and DCC, with lower design complexity and overheads. In general, memory intensive workloads benefit the most from larger cache capacity and compression. YACC achieves on average 12% shorter runtime for high memory intensive workloads, such as apache and zeus. On the other hand, it achieves the lowest benefit (on average 2% better performance) for low and medium memory intensive workloads, such as equake and wupwise.

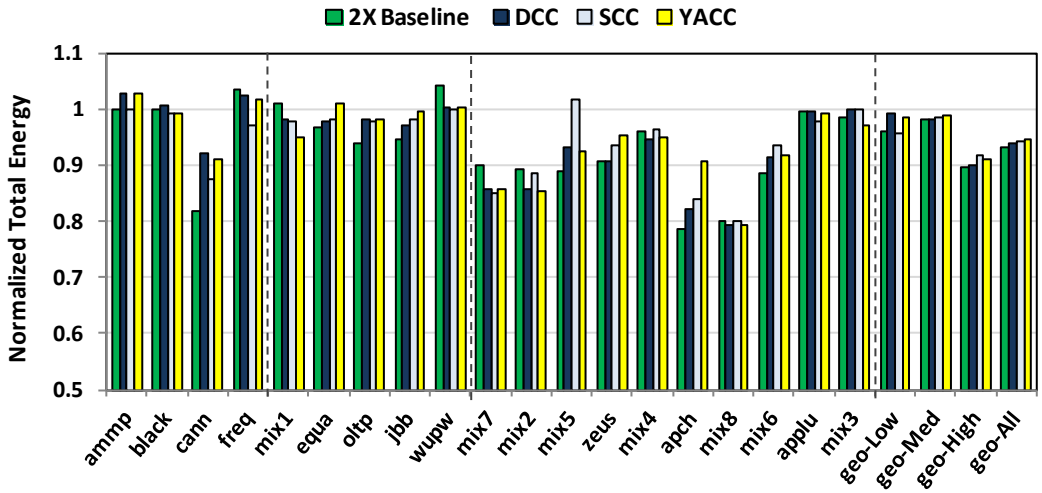

Fig. 6. Normalized total energy

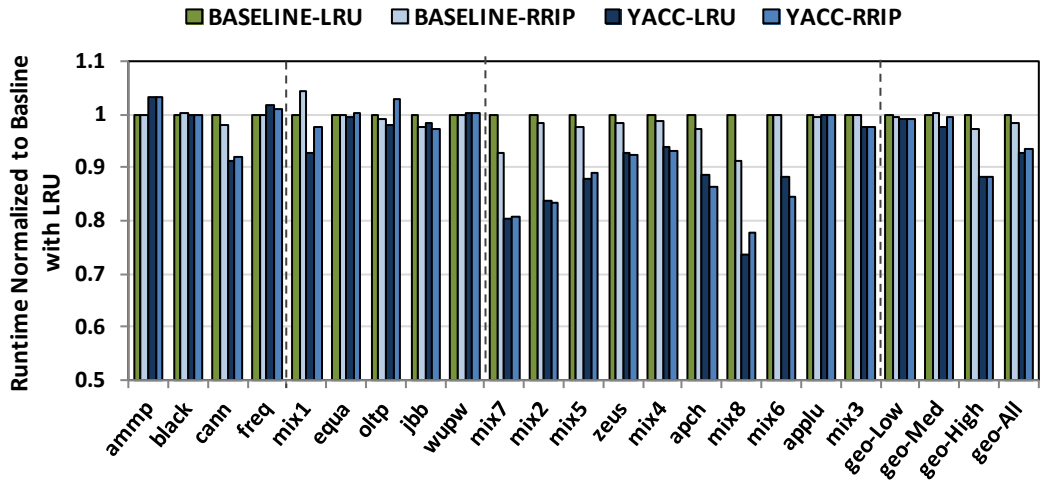

Fig. 7. YACC performance using RRIP

YACC achieves similar performance and energy benefits as DCC and SCC, but YACC has a simpler design that can easily employ any replacement policy. In Figure 7, we illustrate the performance of YACC when using RRIP replacement policy [Jaleel et al. 2010]. For this experiment we use static RRIP, and use 2 bits per super-block tag to store 4 possible re-reference prediction values (RRPV). When allocating a super-block tag for the first time, we set its RRPV to '2' (3-1). On a hit, we promote that super-block tag by setting its RRPV to '0'. When replacing, YACC-RRIP would pick the super-block tag with RRPV of '3'. We also experimented YACC-RRIP with 3-bit RRPV (8 levels) as well. In addition, we considered promoting a super-block tag when inserting a block to that data entry. However, those configurations performed similar to what we presented here. For Baseline-RRIP, we use a similar configuration. We use 2-bit RRPV per block, and promote on cache hits.
As shown in Figure 7, when using RRIP, on average YACC (YACC-RRIP) performs similar to using LRU (YACC-LRU). A regular uncompressed cache with RRIP replacement policy (Baseline-RRIP) also performs on average similar to Baseline-LRU, improving performance for some workloads (e.g., zeus), while lowering or not impacting performance for others (e.g., mix1 and ammp). Overall, this experiment shows that YACC can use alternative replacement policies, while providing its benefits from compression.

CONCLUSIONS

In the few past years, several proposals have addressed many issues preventing the effective hardware implementation of compressed caches.
Our previous proposals, DCC and SCC, reduce the extra hardware complexity induced for storing and retrieving compressed data blocks. In practice with SCC [Sardashti et al. 2014], we addressed most of the issues of the compaction in caches: very limited tag and metadata overhead, direct tag-data matching, no need for defragmentation. However, this comes at the cost of using skewing that induces using one tag array per cache way and has not been widely adopted by industry. In this paper, we introduce Yet Another Compressed Cache (YACC) that retains the same qualities as SCC and addresses the remaining difficulties.
YACC is a simple hardware compressed cache design that achieves the high benefits of previous proposal with low complexity. YACC compacts compressed blocks of a super-block in a single data entry if they can fit. It then uses super-block tags to track them. YACC uses a simple tag-data mapping, storing a block in a set indexed by super-block index independent of its compressibility. In this way, YACC addresses the remaining extra hardware overheads in SCC imposed by skewing, namely complex tag array layout, hashing function hardware, and limited choice for replacement policy. Our simulation experiments show that YACC achieves performance and energy benefits comparable to that of a conventional cache with twice the capacity, and previous work DCC and SCC. However, YACC does this with lower complexity (no skewing) and very limited storage overheads (only 8 extra tag bits per 64 bytes of storage).

As the previous proposals SCC and DCC, YACC can use any possible compression scheme. However as the granularity of cache allocation for both YACC and SCC is 64 bytes, there is a call for the definition of new compression schemes targeting efficient use of these 64 bytes, i.e., compressing two or four contiguous cache lines in 64 bytes.

REFERENCES

A. Alameldeen, and D. Wood. 2004. Adaptive Cache Compression for High-Performance Processors. In Proceedings of the 31st Annual International Symposium on Computer Architecture, 2004.

A. Seznec. 1993. A case for two-way skewed-associative caches. In Proc. of the 20th annual Intl. Symp. on Computer Architecture, 1993.

A. Seznec. 2004. Concurrent Support of Multiple Page Sizes on a Skewed Associative TLB. IEEE Transactions on Computers, 2004.

A. Seznec, F. Bodin. 1993. Skewed-associative caches. Proceedings of PARLE' 93, Munich, June 1993, also available as INRIA Research Report 1655. http://hal.inria.fr/docs/00/07/49/02/PDF/RR-1655.pdf.

Alaa R. Alameldeen, Milo M. K. Martin, Carl J. Mauer, Kevin E. Moore, Min Xu, Mark D. Hill, David A. Wood, Daniel J. Sorin. 2003. Simulating a $2M Commercial Server on a $2K PC, IEEE Computer, 2003.

A. Alameldeen, D. Wood. 2003. Variability in Architectural Simulations of Multi-threaded Workloads, In Proceedings of the Ninth IEEE Symposium on High-Performance Computer Architecture, 2003.

Andre Seznec. 1994. “Decoupled sectored caches: Conciliating low tag implementation cost and low miss ratio,” International Symposium on Computer Architecture, 1994.

Angelos Arelakis, Per Stenstrom. 2014. SC2: A statistical compression cache scheme, In Proceedings of the 41st Annual International Symposium on Computer Architecture, 2014.

Bulent Abali, Hubertus Franke, Xiaowei Shen, Dan E. Poff, and T. Basil Smith. 2001. Performance of Hardware Compressed Main Memory, In Proceedings of the 7th IEEE Symposium on High-Performance Computer Architecture, 2001.

CACTI: http://www.hpl.hp.com/research/cacti/

Christian Bienia and Kai Li. 2009. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors, In Workshop on Modeling, Benchmarking and Simulation, 2009.

Daniel Sanchez, Christos Kozyrakis. 2010. The ZCache: Decoupling Ways and Associativity, in Proceedings of the 43rd annual IEEE/ACM international symposium on Microarchitecture, 2010.

E. Hallnor, S. Reinhardt. 2005. A Unified Compressed Memory Hierarchy, In Proceedings of the 11th International Symposium on High-Performance Computer Architecture, 2005.

Gennady Pekhimenko, Vivek Seshadri, Yoongu Kim, Hongyi Xin, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, Todd C. Mowry. 2013. Linearly Compressed Pages: A Low-Complexity, Low-Latency Main Memory Compression Framework, Annual IEEE/ACM International Symposium on Microarchitecture, 2013.

Intel Core i7 Processors: http://www.intel.com/products/processor/corei7/

Jang-Soo Lee, Won-Kee Hong, Shin-Dug Kim. 2000. An on-chip cache compression technique to reduce decompression overhead and design complexity, Journal of Systems Architecture, 2000.

Jason Zebchuk, Elham Safi, and Andreas Moshovos. 2007. A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy, In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007.

Jeffrey Rothman and Alan Smith. 1999. The Pool of Subsectors Cache Design, International Conference on Supercomputing, 1999.

Julien Dusser, Andre Seznec. 2011. Decoupled Zero-Compressed Memory, In Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, 2011.

Julien Dusser, Thomas Piquet, André Seznec. 2009. Zero-content augmented caches, In Proceedings of the 23rd international conference on Supercomputing, 2009.

Jun Yang and Rajiv Gupta. 2002. Frequent Value Locality and its Applications, ACM Transactions on Embedded Computing Systems, 2002.

Luis Villa, Michael Zhang, and Krste Asanovic. 2000. Dynamic zero compression for cache energy reduction, In Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture, 2000.

M. Ekman, P. Stenstrom. 2005. A robust main-memory compression scheme, SIGARCH Computer Architecture News, 2005.

M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. 2005. Multifacet’s General Execution-driven Multiprocessor Simulator (GEMS) Toolset, Computer Architecture News, 2005.

Nam Sung Kim, Todd Austin, Trevor Mudge. 2002. Low-Energy Data Cache Using Sign Compression and Cache Line Bisection, Second Annual workshop on Memory Performance Issues, 2002.

Somayeh Sardashti and David A. Wood. 2013. Decoupled Compressed Cache: Exploiting Spatial Locality for Energy-Optimized Compressed Caching, Annual IEEE/ACM International Symposium on Microarchitecture, 2013.

Seungcheol Baek, Hyung Gyu Lee, Chrysostomos Nicopoulos, Junghee Lee, and Jongman Kim. 2013. ECM: Effective Capacity Maximizer for High-Performance Compressed Caching, In Proceedings of IEEE Symposium on High-Performance Computer Architecture, 2013.

Somayeh Sardashti and David A. Wood. 2013. Decoupled Compressed Cache: Exploiting Spatial Locality for Energy Optimization, in IEEE Micro Top Picks from the 2013 Computer Architecture Conferences.

S. Sardashti, A. Seznec, and D. Wood. 2014. Skewed Compressed Caches, in the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47), 2014.

Soontae Kim, Jesung Kim, Jongmin Lee, Seokin Hong. 2011. Residue Cache: A Low-Energy Low-Area L2 Cache Architecture via Compression and Partial Hits, In Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, 2011.

Vishal Aslot, M. Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. 2001. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance, In Workshop on OpenMP Applications and Tools, 2001.

Xi Chen, Lei Yang, Robert P. Dick, Li Shang, and Haris Lekatsas. 2010. C-pack: a high-performance microprocessor cache

compression algorithm, IEEE Transactions on VLSI Systems, 2010.
Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., Joel Emer. 2010. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP), ISCA 2010.
Jacob Ziv and Abraham Lempel. 1977. A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory, 23(3):337–343, May 1977.
Jacob Ziv and Abraham Lempel. 1978. Compression of Individual Sequences Via Variable-Rate Coding. IEEE Transactions on Information Theory, 24(5):530 –536, September 1978.
David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. Proc. Inst. Radio Engineers, 40(9):1098–1101, September 1952.
Jeffrey Scott Vitter. 1987. Design and Analysis of Dynamic Huffman Codes. Journal of the ACM, 34(4):825–845, October 1987.
Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. Base-delta-immediate compression: practical data compression for on-chip caches. In Proceedings of the 21st international conference on Parallel architectures and compilation techniques (PACT '12). ACM, New York, NY, USA, 377-388.

RESEARCH CENTRE
BRETAGNE ATALANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex France

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr
ISSN 0249-6399